Cassandra: How it works and what it's good for!

Christopher Batey
Technical Evangelist for Apache Cassandra
@chbatey
Who am I?

• Technical Evangelist for Apache Cassandra
• Founder of Stubbed Cassandra
• Help out Apache Cassandra users
• DataStax
  • Builds enterprise ready version of Apache Cassandra
• Previous: Cassandra backed apps at BSkyB
Overview

• Distributed databases: What and why?
• Cassandra use cases
• Replication
• Fault tolerance
• Read and write path
• Data modelling
• Java Driver
Distributed databases
It is a big world

• Relational
  - Oracle, PostgreSQL

• Graph databases
  - Neo4J, InfoGrid, Titan

• Key value
  - DynamoDB

• Document stores
  - MongoDB, Couchbase

• Columnar aka wide row
  - Cassandra, HBase
Building a web app
Running multiple copies of your app
Still in one DC?
Handling hardware failure
Handling hardware failure
Master/slave

- Master serves all writes
- Read from master and optionally slaves
Peer-to-Peer

- No master
- Read/write to any
- Consistency?
Decisions decisions… CAP theorem

Relational Database

Highly Available Databases: Voldemort, Cassandra

Are these really that different??

Mongo, Redis

@chbatey
Cassandra use cases
Cassandra for Applications

APACHE CASSANDRA
Common use cases

• Ordered data such as time series
  - Event stores
  - Financial transactions
  - Sensor data e.g IoT
Common use cases

• Ordered data such as time series
  - Event stores
  - Financial transactions
  - Sensor data e.g. IoT
• Non functional requirements:
  - Linear scalability
  - High throughout durable writes
  - Multi datacenter including active-active
  - Analytics without ETL
Cassandra deep dive
Cassandra

- Distributed masterless database (Dynamo)
- Column family data model (Google BigTable)
Datacenter and rack aware

- Distributed master less database (Dynamo)
- Column family data model (Google BigTable)
- Multi data centre replication built in from the start
Cassandra

- Distributed master less database (Dynamo)
- Column family data model (Google BigTable)
- Multi data centre replication built in from the start
- Analytics with Apache Spark
Dynamo: Amazon’s Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall
and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon’s core services use to provide an “always-on” experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

One of the lessons our organization has learned from operating Amazon’s platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon’s software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.
Dynamo 101

• The parts Cassandra took
  - Consistent hashing
  - Replication
  - Strategies for replication
  - Gossip
  - Hinted handoff
  - Anti-entropy repair

• And the parts it left behind
  - Key/Value
  - Vector clocks
Picking the right nodes

• You don’t want a full table scan on a 1000 node cluster!
• Dynamo to the rescue: Consistent Hashing
• Then the replication strategy takes over:
  - Network topology
  - Simple
Murmer3 Example

• Data:

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Car</th>
<th>Gender</th>
</tr>
</thead>
<tbody>
<tr>
<td>jim</td>
<td>36</td>
<td>ford</td>
<td>M</td>
</tr>
<tr>
<td>carol</td>
<td>37</td>
<td>bmw</td>
<td>F</td>
</tr>
<tr>
<td>johnny</td>
<td>12</td>
<td></td>
<td>M</td>
</tr>
<tr>
<td>suzy</td>
<td>10</td>
<td></td>
<td>F</td>
</tr>
</tbody>
</table>

• Murmer3 Hash Values:

<table>
<thead>
<tr>
<th>Primary Key</th>
<th>Murmur3 hash value</th>
</tr>
</thead>
<tbody>
<tr>
<td>jim</td>
<td>350</td>
</tr>
<tr>
<td>carol</td>
<td>998</td>
</tr>
<tr>
<td>johnny</td>
<td>50</td>
</tr>
<tr>
<td>suzy</td>
<td>600</td>
</tr>
</tbody>
</table>

Real hash range: -9223372036854775808 to 9223372036854775807
# Murmer3 Example

Four node cluster:

<table>
<thead>
<tr>
<th>Node</th>
<th>Murmur3 start range</th>
<th>Murmur3 end range</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>249</td>
</tr>
<tr>
<td>B</td>
<td>250</td>
<td>499</td>
</tr>
<tr>
<td>C</td>
<td>500</td>
<td>749</td>
</tr>
<tr>
<td>D</td>
<td>750</td>
<td>999</td>
</tr>
</tbody>
</table>
Pictures are better
# Murmer3 Example

Data is distributed as:

<table>
<thead>
<tr>
<th>Node</th>
<th>Start range</th>
<th>End range</th>
<th>Primary key</th>
<th>Hash value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>249</td>
<td>johnny</td>
<td>50</td>
</tr>
<tr>
<td>B</td>
<td>250</td>
<td>499</td>
<td>jim</td>
<td>350</td>
</tr>
<tr>
<td>C</td>
<td>500</td>
<td>749</td>
<td>suzy</td>
<td>600</td>
</tr>
<tr>
<td>D</td>
<td>750</td>
<td>999</td>
<td>carol</td>
<td>998</td>
</tr>
</tbody>
</table>
Replication
Replication strategy

• Simple
  - Give it to the next node in the ring
  - Don’t use this in production

• Network Topology
  - Every Cassandra node knows its DC and Rack
  - Replicas won’t be put on the same rack unless Replication Factor > # of racks
  - Unfortunately Cassandra can’t create servers and racks on the fly to fix this :(
Replication

We have replication!

@chbatey
ALL NODES ARE EQUAL
Tunable Consistency

• Data is replicated N times
• Every query that you execute you give a consistency
  - ALL
  - QUORUM
  - LOCAL_QUORUM
  - ONE

• Christos Kalantzis Eventual Consistency != Hopeful Consistency: http://youtu.be/A6qzx_HE3EU?list=PLqcm6qE9lgKJzVvwH prow9h7KMpb5hcUU
Handling hardware failure
Load balancing

- Data centre aware policy
- Token aware policy
- Latency aware policy
- Whitelist policy
Handling hardware failure

Async Replication

@chbatey
Handling hardware failure

Async Replication

@chbatey
Handling hardware failure

Async Replication
But what happens when they come back?

• Hinted handoff to the rescue
• Coordinators keep writes for downed nodes for a configurable amount of time, default 3 hours
• Longer than that run a repair
Anti entropy repair

• Not exciting but mandatory :)
• New in 2.1 - incremental repair ← awesome
Don’t forget to be social

• Each node talks to a few of its other and shares information

Did you hear node 1 was down???
Scaling shouldn’t be hard

- Throw more nodes at a cluster
- Bootstrapping + joining the ring
  - For large data sets this can take some time

Scale-Up Linearity

Client Writes/s by node count – Replication Factor = 3

@chbatey
Data modelling
You must denormalise
CQL

- Cassandra Query Language
  - SQL like query language
- Keyspace – analogous to a schema
  - The keyspace determines the RF (replication factor)
- Table – looks like a SQL Table

```
CREATE TABLE scores (name text, score int, date timestamp, PRIMARY KEY (name, score));
INSERT INTO scores (name, score, date) VALUES ('bob', 42, '2012-06-24');
INSERT INTO scores (name, score, date) VALUES ('bob', 47, '2012-06-25');
SELECT date, score FROM scores WHERE name='bob' AND score >= 40;
```
Lots of types

<table>
<thead>
<tr>
<th>CQL Type</th>
<th>Constants</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ascii</td>
<td>strings</td>
<td>US-ASCII character string</td>
</tr>
<tr>
<td>bigint</td>
<td>integers</td>
<td>64-bit signed long</td>
</tr>
<tr>
<td>blob</td>
<td>blobs</td>
<td>Arbitrary bytes (no validation), expressed as hexadecimal</td>
</tr>
<tr>
<td>boolean</td>
<td>booleans</td>
<td>true or false</td>
</tr>
<tr>
<td>counter</td>
<td>integers</td>
<td>Distributed counter value (64-bit long)</td>
</tr>
<tr>
<td>decimal</td>
<td>integers, floats</td>
<td>Variable-precision decimal</td>
</tr>
<tr>
<td>double</td>
<td>integers</td>
<td>64-bit IEEE-754 floating point</td>
</tr>
<tr>
<td>float</td>
<td>integers, floats</td>
<td>32-bit IEEE-754 floating point</td>
</tr>
<tr>
<td>inet</td>
<td>strings</td>
<td>IP address string in IPv4 or IPv6 format*</td>
</tr>
<tr>
<td>int</td>
<td>integers</td>
<td>32-bit signed integer</td>
</tr>
<tr>
<td>list</td>
<td>n/a</td>
<td>A collection of one or more ordered elements</td>
</tr>
<tr>
<td>map</td>
<td>n/a</td>
<td>A JSON-style array of literals: { literal : literal, literal : literal ... }</td>
</tr>
<tr>
<td>set</td>
<td>n/a</td>
<td>A collection of one or more elements</td>
</tr>
<tr>
<td>text</td>
<td>strings</td>
<td>UTF-8 encoded string</td>
</tr>
<tr>
<td>timestamp</td>
<td>integers, strings</td>
<td>Date plus time, encoded as 8 bytes since epoch</td>
</tr>
<tr>
<td>uuid</td>
<td>uuids</td>
<td>A UUID in standard UUID format</td>
</tr>
<tr>
<td>timeuuid</td>
<td>uuids</td>
<td>Type 1 UUID only (CQL 3)</td>
</tr>
<tr>
<td>varchar</td>
<td>strings</td>
<td>UTF-8 encoded string</td>
</tr>
<tr>
<td>varint</td>
<td>integers</td>
<td>Arbitrary-precision integer</td>
</tr>
</tbody>
</table>
UUID

• Universal Unique ID
  - 128 bit number represented in character form e.g. 99051fe9-6a9c-46c2-b949-38ef78858dd0
• Easily generated on the client
  - Version 1 has a timestamp component (TIMEUUID)
  - Version 4 has no timestamp component
**TIMEUUID**

TIMEUUID data type supports Version 1 UUIDs
Generated using time (60 bits), a clock sequence number (14 bits), and MAC address (48 bits)

- CQL function `now()` generates a new TIMEUUID

Time can be extracted from TIMEUUID

- CQL function `dateOf()` extracts the timestamp as a date

TIMEUUID values in clustering columns or in column names are ordered based on time

- **DESC** order on TIMEUUID lists most recent data first
CREATE TABLE videos (  
  videoid uuid,  
  userid uuid,  
  name varchar,  
  description varchar,  
  location text,  
  location_type int,  
  preview_thumbnails map<text,text>,  
  tags set<varchar>,  
  added_date timestamp,  
  PRIMARY KEY (videoid)  
);
Data Model - User Defined Types

CREATE TYPE address (  
  street text,  
  city text,  
  zip_code int,  
  country text,  
  cross_streets set<text>  
);

• Complex data in one place  
• No multi-gets (multi-partitions)  
• Nesting!
• We can embed `video_metadata` in `videos`

```
CREATE TABLE videos (  
  videoid uuid,  
  userid uuid,  
  name varchar,  
  description varchar,  
  location text,  
  location_type int,  
  preview_thumbnails map<text,text>,  
  tags set<varchar>,  
  metadata set<frozen<video_metadata>>,  
  added_date timestamp,  
  PRIMARY KEY (videoid)  
);
```

```
CREATE TYPE video_metadata (  
  height int,  
  width int,  
  video_bit_rate set<text>,  
  encoding text  
);
```
Data Model - Storing JSON

```json
{
    "productId": 2,
    "name": "Kitchen Table",
    "price": 249.99,
    "description": "Rectangular table with oak finish",
    "dimensions": {
        "units": "inches",
        "length": 50.0,
        "width": 66.0,
        "height": 32
    },
    "categories": {
        "category": "Home Furnishings",
        "catalogPage": 45,
        "url": "/home/furnishings"
    },
    {"category": "Kitchen Furnishings",
        "catalogPage": 108,
        "url": "/kitchen/furnishings"
    }
}
```

CREATE TYPE dimensions (units text, length float, width float, height float);

CREATE TYPE category (catalogPage int, url text);

CREATE TABLE product (productId int, name text, price float, description text, dimensions frozen <dimensions>, categories map <text, frozen <category>>, PRIMARY KEY (productId));
Tuple type

CREATE TABLE tuple_table (  
id int PRIMARY KEY,  
three_tuple frozen <tuple<int, text, float>>,  
four_tuple frozen <tuple<int, text, float, inet>>,  
five_tuple frozen <tuple<int, text, float, inet, ascii>>  
);

- Type to represent a group
- Up to 256 different elements
Counters

- Old has been around since .8
- Commit log replay changes counters
- Repair can change a counter
Time-to-Live (TTL)

**TTL a row:**
INSERT INTO users (id, first, last) VALUES (‘abc123’, ‘catherine’, ‘cachart’) USING TTL 3600; // Expires data in one hour

**TTL a column:**
UPDATE users USING TTL 30 SET last = ‘miller’ WHERE id = ‘abc123’

- TTL in seconds
- **Can also set default TTL at a table level**
- Expired columns/values automatically deleted
- With no TTL specified, columns/values never expire
- TTL is useful for automatic deletion
- Re-inserting the same row before it expires will overwrite TTL
Example Time: Customer event store
An example: Customer event store

- **Customer event**
  - `customer_id` e.g. ChrisBatey
  - `event_type` e.g. login, logout, add_to_basket, remove_from_basket, buy_item

- **Staff**
  - `name` e.g. Charlie
  - `favourite_colour` e.g. red

- **Store**
  - `name`
  - `type` e.g. Website, PhoneApp, Phone, Retail
Requirements

• Get all events
• Get all events for a particular customer
• As above for a time slice
Modelling in a relational database

CREATE TABLE customer_events(
    customer_id text,
    staff_name text,
    time timeuuid,
    event_type text,
    store_name text,
    PRIMARY KEY (customer_id));

CREATE TABLE store(
    name text,
    location text,
    store_type text,
    PRIMARY KEY (store_name));

CREATE TABLE staff(
    name text,
    favourite_colour text,
    job_title text,
    PRIMARY KEY (name));
Your model should look like your queries
Modelling in Cassandra

CREATE TABLE customer_events(
  customer_id text,
  staff_id text,
  time timeuuid,
  store_type text,
  event_type text,
  tags map<text, text>,
  PRIMARY KEY ((customer_id), time));

CQLSH: customers> select * from customer_events where customer_id = 'chbatey' and time > minTimeuuid(1) and time < maxTimeuuid(200000000000);

customer_id | time         | event_type | staff_id | store_type | tags
--------------|--------------|------------|----------|------------|------------------
chbatey       | 2b329cc0-73f0-11e4-ac06-4b05b98cc84c | basket_add | trevor   | online     | {'item': 'coffee'}
chbatey       | 6a823160-73f0-11e4-ac06-4b05b98cc84c | basket_add | trevor   | online     | {'item': 'coffee'}
### How it is stored on disk

<table>
<thead>
<tr>
<th>customer</th>
<th>time</th>
<th>event_type</th>
<th>store_type</th>
<th>tags</th>
</tr>
</thead>
<tbody>
<tr>
<td>charles</td>
<td>2014-11-18 16:52:04</td>
<td>basket_add</td>
<td>online</td>
<td>{'item': 'coffee'}</td>
</tr>
<tr>
<td>charles</td>
<td>2014-11-18 16:53:00</td>
<td>basket_add</td>
<td>online</td>
<td>{'item': 'wine'}</td>
</tr>
<tr>
<td>charles</td>
<td>2014-11-18 16:53:09</td>
<td>logout</td>
<td>online</td>
<td>{}</td>
</tr>
<tr>
<td>chbatey</td>
<td>2014-11-18 16:52:21</td>
<td>login</td>
<td>online</td>
<td>{}</td>
</tr>
<tr>
<td>chbatey</td>
<td>2014-11-18 16:53:21</td>
<td>basket_add</td>
<td>online</td>
<td>{'item': 'coffee'}</td>
</tr>
<tr>
<td>chbatey</td>
<td>2014-11-18 16:54:00</td>
<td>basket_add</td>
<td>online</td>
<td>{'item': 'cheese'}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>charles</th>
<th>event_type</th>
<th>staff_id</th>
<th>store_type</th>
<th>tags: item</th>
<th>event_type</th>
<th>staff_id</th>
<th>store_type</th>
<th>tags: item</th>
<th>event_type</th>
<th>staff_id</th>
<th>store_type</th>
<th>tags: item</th>
</tr>
</thead>
<tbody>
<tr>
<td>basket_add</td>
<td>n/a</td>
<td>online</td>
<td>coffee</td>
<td></td>
<td>basket_add</td>
<td>n/a</td>
<td>online</td>
<td></td>
<td>logout</td>
<td>n/a</td>
<td>online</td>
<td>cheese</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>chbatey</th>
<th>event_type</th>
<th>staff_id</th>
<th>store_type</th>
<th>event_type</th>
<th>staff_id</th>
<th>store_type</th>
<th>tags: item</th>
<th>event_type</th>
<th>staff_id</th>
<th>store_type</th>
<th>tags: item</th>
</tr>
</thead>
<tbody>
<tr>
<td>login</td>
<td>n/a</td>
<td>online</td>
<td></td>
<td></td>
<td>n/a</td>
<td>online</td>
<td></td>
<td>camera</td>
<td>n/a</td>
<td>online</td>
<td>cheese</td>
</tr>
</tbody>
</table>
Drivers
Languages

• DataStax (open source)
  - C#, Java, C++, Python, Node, Ruby
  - Very similar programming API

• Other open source
  - Go
  - Clojure
  - Erlang
  - Haskell
  - Many more Java/Python drivers
  - Perl
DataStax Java Driver

• Open source

DataStax Java Driver for Apache Cassandra

A Java client driver for Apache Cassandra. This driver works exclusively with the Cassandra Query Language version 3 (CQL3) and Cassandra’s binary protocol.

- JIRA: https://datastax-oss.atlassian.net/browse/JAVA
- MAILING LIST: https://groups.google.com/a/lists.datastax.com/forum/#!forum/java-driver-user
- IRC: #datastax-drivers on irc.freenode.net
- TWITTER: Follow the latest news about DataStax Drivers - @olim7t, @mfiguierer
- API: http://www.datastax.com/drivers/java/2.1
- CHANGELOG: https://github.com/datastax/java-driver/blob/2.1/driver-core/CHANGELOG.rst

The driver architecture is based on layers. At the bottom lies the driver core. This core handles everything related to the connections to a Cassandra cluster (for example, connection pool, discovering new nodes, etc.) and exposes a simple, relatively low-level API on top of which higher level layer can be built.

<dependency>
  <groupId>com.datastax.cassandra</groupId>
  <artifactId>cassandra-driver-core</artifactId>
  <version>2.1.2</version>
</dependency>

<dependency>
  <groupId>com.datastax.cassandra</groupId>
  <artifactId>cassandra-driver-mapping</artifactId>
  <version>2.1.2</version>
</dependency>
public List<CustomerEvent> getAllCustomerEvents() {
    return session.execute("select * from customers.customer_events")
        .all().stream()
        .map(mapCustomerEvent())
        .collect(Collectors.toList());
}

private Function<Row, CustomerEvent> mapCustomerEvent() {
    return row -> new CustomerEvent(
        row.getString("customer_id"),
        row.getUUID("time"),
        row.getString("staff_id"),
        row.getString("store_type"),
        row.getString("event_type"),
        row.getMap("tags", String.class, String.class));
}
All events for a particular customer

```java
private PreparedStatement getEventsForCustomer;

@PostConstruct
public void prepareStatements() {
    getEventsForCustomer = 
        session.prepare("select * from customers.customer_events where customer_id = ?");
}

public List<CustomerEvent> getCustomerEvents(String customerId) {
    BoundStatement boundStatement = getEventsForCustomer.bind(customerId);
    return session.execute(boundStatement)
        .all().stream()
        .map(mapCustomerEvent())
        .collect(Collectors.toList());
}
```
public List<CustomerEvent> getCustomerEventsForTime(String customerId, long startTime, long endTime) {

    Select.Where getCustomers = QueryBuilder.select()
        .all()
        .from("customers", "customer_events")
        .where(eq("customer_id", customerId))
        .and(gt("time", UUIDs.startOf(startTime)))
        .and(lt("time", UUIDs.endOf(endTime)));

    return session.execute(getCustomers).all().stream()
        .map(mapCustomerEvent())
        .collect(Collectors.toList());
}
Mapping API

```java
@Table(keyspace = "customers", name = "customer_events")
public class CustomerEvent {
    @PartitionKey
    @Column(name = "customer_id")
    private String customerId;

    @ClusteringColumn
    private UUID time;

    @Column(name = "staff_id")
    private String staffId;

    @Column(name = "store_type")
    private String storeType;

    @Column(name = "event_type")
    private String eventType;

    private Map<String, String> tags;
    // ctr / getters etc
}
```
Mapping API

@Accessor
public interface CustomerEventDao {
    @Query("select * from customers.customer_events where customer_id = :customerId")
    Result<CustomerEvent> getCustomerEvents(String customerId);

    @Query("select * from customers.customer_events")
    Result<CustomerEvent> getAllCustomerEvents();

    @Query("select * from customers.customer_events where customer_id = :customerId and time > minTimeuuid(:startTime) and time < maxTimeuuid(:endTime)")
    Result<CustomerEvent> getCustomerEventsForTime(String customerId, long startTime, long endTime);
}

@Bean
public CustomerEventDao customerEventDao() {
    MappingManager mappingManager = new MappingManager(session);
    return mappingManager.createAccessor(CustomerEventDao.class);
}
Adding some type safety

```java
public enum StoreType {
    ONLINE, RETAIL, FRANCHISE, MOBILE
}

@Table(keyspace = "customers", name = "customer_events")
public class CustomerEvent {
    @PartitionKey
    @Column(name = "customer_id")
    private String customerId;

    @ClusteringColumn()
    private UUID time;

    @Column(name = "staff_id")
    private String staffId;

    @Column(name = "store_type")
    @Enumerated(EnumType.STRING) // could be EnumType.ORDINAL
    private StoreType storeType;
}
```
User defined types

create TYPE store (name text, type text, postcode text) ;

CREATE TABLE customer_events_type(
customer_id text,
staff_id text,
time timeuuid,
store frozen<store>,
event_type text,
tags map<text, text>,
PRIMARY KEY ((customer_id), time));
Mapping user defined types

@UDT(keyspace = "customers", name = "store")
public class Store {
    private String name;
    private StoreType type;
    private String postcode;
    // getters etc
}

@Table(keyspace = "customers", name = "customer_events_type")
public class CustomerEventType {
    @PartitionKey
    @Column(name = "customer_id")
    private String customerId;

    @ClusteringColumn()
    private UUID time;

    @Column(name = "staff_id")
    private String staffId;

    @Frozen
    private Store store;

    @Column(name = "event_type")
    private String eventType;

    private Map<String, String> tags;
Mapping user defined types

```java
@UDT(keyspace = "customers", name = "store")
public class Store {
    private String name;
    private StoreType type;
    private String postcode;
    // getters etc
}

@Table(keyspace = "customers", name = "customer_events_type")
public class CustomerEventType {
    @PartitionKey
    @Column(name = "customer_id")
    private String customerId;

    @ClusteringColumn()
    private UUID time;

    @Column(name = "staff_id")
    private String staffId;

    @Frozen
    private Store store;

    @Column(name = "event_type")
    private String eventType;

    private Map<String, String> tags;

    @Query("select * from customers.customer_events_type")
    Result<CustomerEventType> getAllCustomerEventsWithStoreType();
```
Other features
# Query Tracing

Connected to cluster: xerxes
Simplex keyspace and schema created.
Host (queried): /127.0.0.1
Host (tried): /127.0.0.1
Trace id: 96ac9400-a3a5-11e2-96a9-4db56cdc5fe7

<table>
<thead>
<tr>
<th>activity</th>
<th>timestamp</th>
<th>source</th>
<th>source_elapsed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parsing statement</td>
<td>12:17:16.736</td>
<td>/127.0.0.1</td>
<td>28</td>
</tr>
<tr>
<td>Preparing statement</td>
<td>12:17:16.736</td>
<td>/127.0.0.1</td>
<td>199</td>
</tr>
<tr>
<td>Determining replicas for mutation</td>
<td>12:17:16.736</td>
<td>/127.0.0.1</td>
<td>348</td>
</tr>
<tr>
<td>Sending message to /127.0.0.3</td>
<td>12:17:16.736</td>
<td>/127.0.0.1</td>
<td>788</td>
</tr>
<tr>
<td>Sending message to /127.0.0.2</td>
<td>12:17:16.736</td>
<td>/127.0.0.1</td>
<td>805</td>
</tr>
<tr>
<td>Acquiring switchLock read lock</td>
<td>12:17:16.736</td>
<td>/127.0.0.1</td>
<td>828</td>
</tr>
<tr>
<td>Appending to commitlog</td>
<td>12:17:16.736</td>
<td>/127.0.0.1</td>
<td>848</td>
</tr>
<tr>
<td>Adding to songs memtable</td>
<td>12:17:16.736</td>
<td>/127.0.0.1</td>
<td>900</td>
</tr>
<tr>
<td>Message received from /127.0.0.1</td>
<td>12:17:16.737</td>
<td>/127.0.0.2</td>
<td>34</td>
</tr>
<tr>
<td>Message received from /127.0.0.1</td>
<td>12:17:16.737</td>
<td>/127.0.0.3</td>
<td>25</td>
</tr>
<tr>
<td>Acquiring switchLock read lock</td>
<td>12:17:16.737</td>
<td>/127.0.0.2</td>
<td>672</td>
</tr>
<tr>
<td>Acquiring switchLock read lock</td>
<td>12:17:16.737</td>
<td>/127.0.0.3</td>
<td>525</td>
</tr>
</tbody>
</table>
public void storeEventLogged(CustomerEvent customerEvent) {
    BoundStatement boundInsertForCustomerId = insertByCustomerId.bind(customerEvent.getCustomerId(),
        customerEvent.getTime(),
        customerEvent.getEventType(),
        customerEvent.getStaffId(),
        customerEvent.getStaffId());

    BoundStatement boundInsertForStaffId = insertByStaffId.bind(customerEvent.getCustomerId(),
        customerEvent.getTime(),
        customerEvent.getEventType(),
        customerEvent.getStaffId(),
        customerEvent.getStaffId());

    BatchStatement batchStatement = new BatchStatement(BatchStatement.Type.LOGGED);
    batchStatement.add(boundInsertForCustomerId);
    batchStatement.add(boundInsertForStaffId);

    session.execute(batchStatement);
}
Why is this slower?

client

BL-R: Batch log replica

BATCH LOG
Light weight transactions

- Often referred to as compare and set (CAS)

```sql
INSERT INTO STAFF (login, email, name)
values ('chbatey', 'christopher.batey@datastax.com', 'Chirstopher Batey')
IF NOT EXISTS
```
Summary

• Cassandra is a shared nothing masterless datastore
• Availability a.k.a up time is king
• Biggest hurdle is learning to model differently
• Modern drivers make it easy to work with
Thanks for listening

• Follow me on twitter @chbatey
• Cassandra + Fault tolerance posts a plenty:
  • http://christopher-batey.blogspot.co.uk/
• Cassandra resources: http://planetcassandra.org/
• Full free day of Cassandra talks/training:
  • http://www.eventbrite.com/e/cassandra-day-london-2015-april-22nd-2015-tickets-15053026006?aff=meetup1